

PREFACE

The idea behind this revised effort is to add to already available manual for the *TMS 320C6211 DSK*. The earlier manual was a good effort in terms of the amount of work done and the experiments designed, but it lacked the documentation and commenting of the codes. This is an effort in that direction.

We have tried to study the already available codes. The aim was not to rewrite the codes but to present them in a form that the person reading them would be able to understand what they contain and what is going on the host of register initializations. We have commented the codes and added suggestions as to what other initializations could be done in their place wherever necessary. Our effort is no way full-proof, but we have tried hard to improve upon the existing documentation and present it in a way that is understandable much more easily.

It would be very nice if the people using this manual add to it their own experiences in programming the DSK and help in building a comprehensive reference for the DSK.

Gaurav Verma
Hari Raghavendra
Mohit Garg

Overview of the TMS 320C6211

The *TMS 320C6211 DSK* is a Fixed-Point DSP which is a part of the C62x series of Texas Instruments (TI). A DSK consists of an onboard DSP and peripherals which are connected together on the main PCB. There are ports and sockets for external connections also. It consists of 32-bit long registers.

Here we list down the available peripherals on the board and a few lines on each of them. (Refer to the TI documentation for complete information).

1. **DMA Controller:** This is the Direct Memory Access controller inbuilt on the board. The function is same as a normal DMA controller (transfers data between address ranges in the memory map without intervention by the CPU). It has 4 programmable channels and a 5th auxiliary channel.
2. **EDMA Controller:** The Enhanced Direct Memory Access controller performs the same function as the DMA but is more advanced and has 16 programmable channels.
3. **HPI:** The Host Port Interface, is a parallel port through which a host processor can directly access the CPU's (DSP's) memory space. The host device is made the master of the interface, hence, it has ease of access. The host can directly access the memory mapped devices and peripherals.
4. **EMIF:** The Extended Memory Interface is an interface which can be used for connecting external memory devices to the CPU. *Here by "external", we mean devices external to the CPU memory. i.e. external devices include the on-chip peripherals and anything outside the CPU of the system.*

5. **Expansion Bus:** It can be used as a replacement for the HPI as well as an extension for the EMIF. The host port of the expansion bus can operate in either asynchronous slave mode (similar to HPI), or in synchronous master/slave mode. This allows the device to interface to a variety of host bus protocols.
6. **McBSP:** The Multi-Channel Buffered Serial Port. This document is mainly focused on this peripheral of the DSK owing to the role it plays in the experiments we are studying. It can buffer serial samples in memory with the help of DMA/EDMA controllers. It has a multi-channel transmission and reception capability upto 128 channels. The clocking and framing can be programmed using internal registers.
7. **Interrupt Selector:** The CPU has 12 interrupts available and the interrupt selector allows one to choose which 12 interrupts our system needs.
8. **BOOT Configuration:** This determines which actions the DSP performs after device reset to prepare for initialization. These may include loading a specified code from internal/external memory unit.
9. **Power Down Mode:** This can be used to save power while using the DSK. It can be viewed as some sort of a “standby” mode of the DSK when some or all of the chips can be prevented from switching without losing any data or operational context.(see documentation for details).
10. **Timers:** The DSK has two 32-bit timers on-board.

This was a small overview of the main sections of the DSK. The same task can be performed using more than one peripheral. Most of the experiments which are given in this manual are based on using the McBSP for input/output directly via the CPU. The experiments can also be done by using the EDMA controller.

Programming the DSK

The DSK can be viewed as a microprocessor with lots of inbuilt peripherals attached to it (like the Anshuman Kits of the 8085 system). The peripherals are there for ease of access and are useful in programming the system. Our main aim in this exercise is to program the DSP to take in Analog data, process it and give out the processed Analog data.

Since, the processing cannot be done on Analog systems that easily and the fact that this is a DSP lab (!), we need to sample the input data, process it and send it to a D/A converter. The A/D and D/A conversion in the C6211 is done using the AD535 chip which can be seen near the input and output ports of the board.

This chip is interfaced with the **McBSP** port directly and we will be programming the McBSP and using it to interface with this device. Also, since the codes in general may need more program and memory space, we need to initialize the **EMIF**. This may not be necessary in some of the codes that we try here but it is a good habit to do it as otherwise one may lead to overflow conditions and unexpected behaviour may result.

Before we start off, it would be advisable that you get yourself familiar with the Code Composer Studio (CCS) that comes with the C6211 kit. It simplifies life a lot by allowing one to directly write C code and automatically interfaces it with the DSP. The CCS is very similar to VC++ or any such software. The main idea is to create a new project and add the required files to it. (Refer to the earlier manual for a brief tutorial).

One file that is very important for loading the program into the DSP memory is the "lnk.cmd" file. This file is a sort of mapping between the labels in the

assembled code and the part of the memory where the variables inside these blocks are stored. You may want to have a look at it before you proceed.

Structure of the Program

In any embedded system program, one will have to initialize the required registers before proceeding so as to have a complete control over the system parameters. Here also, the same thing is done.

All the programs that we consider will be having the same structure as the main idea behind all is the same—take in the data, process it and give it back. We also use the same way of reading/writing to the McBSP. We will be polling the control registers to read and write data to the ports.

Header Files

The header files included are standard C header files and two files which tell the compiler about the parameters and register memory addresses (most of the peripheral registers and memory mapped). You can open the files and see for yourself. The “c6x.h” is the standard path of the CCS directory and the “c6211dsk.h” is in the same directory as the source file.

Initializations

The programs start with some register initializations. We go through them one by one.

1. CSR, Control Status Register: (register address: 00001h) This register contains the control and status bits which define the environment in which the program is to be executed. The fields are as follows :

a) Bit Positions 31-24 and 23-16 :

These specify the CPU type and the CPU silicon revision id respectively. We take them as 00h and 00h respectively.

b) Bit Positions 15-10

These control the CPU power down mode. We write 000000b (value after reset).

c) Bit Position 9

This is a Read-Only bit which is set whenever any part of the CPU performs a saturate. Since it is a don't care, we write it as 0b.

d) Bit Position 8

This specifies the "endian" of the system. i.e. little or big endian. Since, we are using the little endian modes, we need to include the "rts6201.lib" file. For big endian mode, one should use the "rts6201e.lib" file.

e) Bit positions 7-5 and 4-2

These specify the Cache for the Program and Data memories. The programs have the caches disabled but one can very well enable them by writing the appropriate bit pattern.(010 for each)

f) Bit Positions 1 and 0

These two positions specify the Interrupt status. Bit 0 is the GIE (Global Interrupt Enable) flip flop, which can be used to mask out all Maskable Interrupts. We set it to 0b to do so. Bit 1 is the PGIE (Previous GIE) which is used to store the value of the GIE when an interrupt is serviced.

Hence the control word becomes 100h. We can also enable the cache and appropriately change the control word.

2. IER , Interrupt Enable Register(memory address: 00100h): This is used for individual interrupt control. One can use the bits in this register to enable/disable individual interrupts. Note that these bits have significance only if the GIE field of the CSR is set i.e. the interrupts are not already masked out by the GIE.

Another point to be kept in mind is that the NMIE field (bit position 1) disables all non-reset interrupts. This field cannot be manually cleared. At reset NMIE=0 to prevent any interruption of the CPU. One must manually set the NMIE after reset to be able to use NMI.

The IER has a few associated registers which can be used to control interrupts more efficiently.

- i. IFR, Interrupt Flag Register (memory address: 00010h): This register stores the status of pending interrupts. This is a Read-Only register.
 - ii. ISR, Interrupt Set Register (memory address: 00010h): This register has the same address as the IFR but is a Write-Only register. This can be used to set the corresponding bit of the IFR.
 - iii. ICR, Interrupt Clear Register (memory address: 00011h): This register can be used to clear the corresponding bits of the IFR.
- The filed descriptions of the ISR and ICR are the same as that of the IER.
 - The bits of IFR can be changed only by writing a '1' at the corresponding positions of the ISR or ICR. Writing a '0' to the positions has not effect whatsoever.
 - NMI or reset cannot be set or cleared by any bit in the ICR or ISR.

In the code we have set the IER =1. We could have as well written a '0' as the first bit is a Read-Only field. Also, we have cleared all pending interrupts by writing a FFFFh in the ICR. This is done because we donot need any interrupts in the program.

The registers that we have seen so far have been initialized as a simple assignment statement, i.e. they have been used as variables. The next set of initializations are done by casting into pointers and then initializing. These have been declared in the "c6211dsk.h" file.

3. EMIF Registers:

These registers are used to define the memories to use in the loading the program on the CPU. The various registers are listed as follows. In order to

alter the values, you should make yourself familiar with the types of memories on the board and their specifications.

- i. GCR, Global Control Register (memory address: 0180 0000h): Specifies the overall control signals and clocks of the memory interfacing cycles.
- ii. CE Space Control registers: There are 4 such registers which control the CE space parameters and memory type for each of the memory spaces. The MTYPE field should be initialized only once during system initialization. If the system selects SDRAM/SBSRAM as the MTYPE, the remaining fields have no effect.
- iii. SDCTRL, SDRAM Control Register (address: 0180 0018h): This controls the SDRAM parameters for the spaces which have SDRAM as the memory type in the corresponding CE Space Control register.
- iv. SDRP (memory address: 0180 001Ch): This controls the SDRAM refresh period in terms of the ECLKOUT pin clock. The number of refreshes can be controlled via the XRFR field.
This can also be used as a general purpose counter if SDRAM is not used by the system. When this counter reaches '0', it is automatically reloaded and an interrupt (SDINT) is sent to the interrupt selector.
- v. SDEXT, SDRAM extension (memory address: 0180 0020h): This offers more control in setting the SDRAM parameters.

4. McBSP Registers:

These registers control the serial port parameters like the clock speed, word length, interrupt specifications and other parameters.

- i. DRR, Data Receive Register: DRR contains the data received from the Receive Buffer Register and transfers it to the CPU or the DMA controller, as the case may be. The DRR is mapped on to the following memory locations :
 1. McBSP 0 : 018C 0000
 2. McBSP 1 : 0190 0000
 3. McBSP 2 : 01A4 0000 (6202 and 6203 only)

For the 6211 and 6711 devices the DRR is also mapped on to the memory locations 3000 0000 - 33FFFFFF (for McBSP 0) and 34000000 - 3FFFFFFF (for McBSP1). Both the CPU and the DMA controller can access the DRR in all these memory

locations. A read from any location in 3000 0000 - 33FFFFFF is equivalent to a read from the DRR of McBSP 0 at 018C 0000. Similarly a read from any location in 34000000 - 3FFFFFFF is equivalent to a read from the DRR of McBSP 1 at 0190 0000.

Hence we have a choice of reading from either 3XXXXXXXX or 018C 0000/0190 0000 location. However accesses to the 018C 0000/0190 0000 locations go through the peripheral bus. Therefore it is recommended that you set up the EDMA to use the 3XXXXXXXX addresses for serial port addressing in order to free up peripheral bus for other functions.

- ii. DXR, Data Transmit Register: DXR contains the data from the CPU or the DMA controller as the case may be, which is to be transmitted to the Data Transmit pin through the Transmit Shift Register. The DXR is mapped on to the following memory locations :

1. McBSP 0 : 018C 0004
2. McBSP 1 : 018C 0004
3. McBSP 2 : 018C 0004 (6202 and 6203 only)

For the 6211 and 6711 devices the DXR is also mapped on to the memory Locations 3000 0000 - 33FFFFFF (for McBSP 0) and 34000000 - 3FFFFFFF (for McBSP 1). Both the CPU and the DMA controller can access the DXR in all these memory locations. A write to any location in 3000 0000 - 33FFFFFF is equivalent to write to the DXR of McBSP 0 at 018C 0004. Similarly a read from any location in 34000000 - 3FFFFFFF is equivalent to a read from the DRR of McBSP 1 at 0190 0004.

Hence we have a choice of reading from either 3XXXXXXXX or 018C 0004/0190 0004 location .However accesses to the 018C 0004/0190 0004 locations go through the peripheral bus. Therefore it is recommended that you set up the EDMA to use the 3XXXXXXXX addresses for serial port addressing in order to free up peripheral bus for other functions

- iii. SPCR, Serial Port Control Register (address: 018C 0008h): This controls the parameters like frame generation, transmitter and receiver parameters, synchronization settings etc. Here we have initially made it '0' in order to reset the McBSP. Later on, we set it to enable receive and transmit data through the RRST and XRST pins. We also make RJUST=010b in order to sign extend the received data. Hence, we set this register to 012001h.

- iv. PCR, Pin Control Register (address: 018C 0024h): This is used to set the clocks for transmit/receive, polarity of the clocks etc. Since, the port in our case is driven by an external clock, we set this register 0.
- v. R(X)CR, Receive (Transit) Control Register (address: 018C 0004h(0010h)): These are used to set the word length, frame length and no. of words in each phase. Companding can also be set using these registers.
- vi. SRGR, Sample Rate Generator Register (address: 018C 0014h): This register is used to set the clock frequency at which the input serial data is sampled. This register is used when we want the clock to be an internal signal and not driven by an external source. We donot use it in our program as we are not using internal clock generation here.
Note that the SRGR clock frequency is not the sampling frequency of the input Analog Signal. The sampling frequency is determined by the A/D converter and the SRGR does not control that. The SRGR frequency determines the frequency at which the input to the receiver (DR pin) is sampled which in our case is the data from the A/D converter.

The Serial Port Initialization procedure is as follows:

1. Disable the receiver and transmitter using XRST and RRST in SPCR.
2. Program the McBSP configuration registers. Donot program the data registers. Note that the configuration registers should be changed only when the affected part of the serial port is in RESET. Also, the data registers should be loaded only when the device is out of RESET e.g. DXR should be loaded only when transmitter is enabled.
3. Wait 2-bit clocks for proper internal synchronization.
4. Set up the data acquisition as desired.
5. Enable the serial ports using XRST and RRST in SPCR.
6. The value written to the SPCR should have only the reset bits changed and the remaining bit fields should have the same value as in step 2.
7. Enable the frame synchronization generator, if required.

Programs

Now that we have studied the initialization of the required registers of the DSK and the McBSP , we shall look at the programs that do particular signal processing. We shall study following examples of signal processing here :

(1) Waveform Generation Experiments

- Generation Of a Square Waveform.
- Generation Of a Sawtooth Waveform.
- Generation Of a Triangular Waveform.

(2) Sampling

- Real time sampling of analog signals

(3) Discrete Time Filtering

- High-pass and Low-pass FIR filters
- High-pass and Low-pass IIR filters

Waveform Generation

Square Waveform Generation Code (These are the original codes reproduced here for convenience, You can try the suggested changes in the initialization part as indicated above).

```
#include <stdio.h>
#include <c6x.h>
#include "c6211dsk.h"
#include "codec_poll.h"

int main()
{

    /* DSP and peripheral initialization */
    CSR=0x100;          /* disable all interrupts */
    IER=1;             /* disable all interrupts except NMI */
    ICR=0xffff;       /* clear all pending interrupts */

    *(unsigned volatile int *)EMIF_GCR = 0x3300; /* EMIF global control*/
    *(unsigned volatile int *)EMIF_CE0 = 0x30; /* EMIF CE0control */
    *(unsigned volatile int *)EMIF_CE1 = 0xffff03;
    /* EMIF CE1 control, 8bit async */

    *(unsigned volatile int *)EMIF_SDCTRL = 0x07117000;
    /* EMIF SDRAM control */
    *(unsigned volatile int *)EMIF_SDRP = 0x61a;
    /* EMIF SDRAM refresh period */
    *(unsigned volatile int *)EMIF_SDEXT = 0x54519;
    /* EMIF SDRAM extension */

    mcbasp0_init();
    codec_playback();

    return(0);
}
```

```

void mcbasp0_init()
{
    /* set up McBSP */

    *(unsigned volatile int *)McBSP0_SPCR = 0;          /* reset serial port */
    *(unsigned volatile int *)McBSP0_PCR = 0;          /* set pin control reg.*/
    *(unsigned volatile int *)McBSP0_RCR = 0x10040;
    /*set rx control reg. one 16 bit data/frame */

    *(unsigned volatile int *)McBSP0_XCR = 0x10040;
    /*set tx control reg. one 16 bit data/frame */
    *(unsigned volatile int *)McBSP0_DXR = 0;
    *(unsigned volatile int *)McBSP0_SPCR = 0x12001; /*setup SP control reg.*/
}

void mcbasp0_write(int out_data)
{
    int temp;
    temp = *(unsigned volatile int *)McBSP0_SPCR & 0x20000;
    while ( temp == 0)
    {
        temp = *(unsigned volatile int *)McBSP0_SPCR & 0x20000;
    }
    *(unsigned volatile int *)McBSP0_DXR = out_data;
}

int mcbasp0_read()
{
    int temp;
    temp = *(unsigned volatile int *)McBSP0_SPCR & 0x2;
    while ( temp == 0)
    {
        temp = *(unsigned volatile int *)McBSP0_SPCR & 0x2;
    }
    temp = *(unsigned volatile int *)McBSP0_DRR;
    return temp;
}

void codec_playback()
{
    int temp, z;

```

```

double x[2], y;

// set up control register 3 for S/W reset
mcbasp0_read();
mcbasp0_write(0);
mcbasp0_read();
mcbasp0_write(0);
mcbasp0_read();
mcbasp0_write(0);
mcbasp0_read();
mcbasp0_write(1);
mcbasp0_read();
mcbasp0_write(0x0386);
mcbasp0_read();
mcbasp0_write(0);
mcbasp0_read();

// set up control register 3 for mic input
mcbasp0_write(0);
mcbasp0_read();
mcbasp0_write(0);
mcbasp0_read();
mcbasp0_write(1);
mcbasp0_read();
mcbasp0_write(0x0306);
mcbasp0_read();
mcbasp0_write(0);
mcbasp0_read();

mcbasp0_write(0);
mcbasp0_read();
mcbasp0_write(1);
mcbasp0_read();
mcbasp0_write(0x2330);
temp = mcbasp0_read();
mcbasp0_write(0x0);
mcbasp0_read();
mcbasp0_write(0x0);
mcbasp0_read();
if((temp & 0xff) != 0x06)
{
#ifdef PRINT
    printf ("Error in setting up register 3.\n");
    exit(0);
#endif
}

```

```

}

//set up control register 4
mcbasp0_write(0);
mcbasp0_read();
mcbasp0_write(0);
mcbasp0_read();
mcbasp0_write(1);
mcbasp0_read();
mcbasp0_write(0x0400);
mcbasp0_read();
mcbasp0_write(0);
mcbasp0_read();

mcbasp0_write(0);
mcbasp0_read();
mcbasp0_write(1);
mcbasp0_read();
mcbasp0_write(0x2430);
temp = mcbasp0_read();
mcbasp0_write(0x0);
mcbasp0_read();
mcbasp0_write(0x0);
mcbasp0_read();
if((temp & 0xff) != 0x00)
{
#if PRINT
    printf("Error in setting up register 4.\n");
    exit(0);
#endif
}

// set up control register 5
mcbasp0_write(0);
mcbasp0_read();
mcbasp0_write(0);
mcbasp0_read();
mcbasp0_write(1);
mcbasp0_read();
mcbasp0_write(0x0502);
mcbasp0_read();
mcbasp0_write(0);
mcbasp0_read();

mcbasp0_write(0);
mcbasp0_read();

```

```
mcbbsp0_write(1);
mcbbsp0_read();
mcbbsp0_write(0x2530);
temp = mcbbsp0_read();
mcbbsp0_write(0x0);
mcbbsp0_read();
mcbbsp0_write(0x0);
mcbbsp0_read();
if((temp & 0xfe) != 0x2)
{
#ifdef PRINT
    printf ("Error in setting up register 5.\n");
    exit(0);
#endif
}
```

```
while(1) /* play back about 5 minutes */
{
    for(i=0;i<100;i++)
    {
        temp = 20000;
        temp = temp & 0xfffe;
        mcbbsp0_write(temp);
    }
    for(i=0;i<100;i++)
    {
        temp = 0;
        temp = temp & 0xfffe;
        mcbbsp0_write(temp);
    }
}
}
```

Some important observations about the above waveform generation code

- In the above code we have used the value 20000 and 0 to generate the maximum and minimum values of voltage in the output square wave. The square wave generated as an output of this code is 1 V p-to-p (measured on the CRO). Hence we observe that the value of 20000 in the code corresponds to a value of 1 V p-to-p in the output signal.
- There is an inversion of the voltages in the output signal. That is, the value of 20000 in the code corresponds to the minimum value of the voltage and the value of 0 corresponds to the maximum value of the voltage in the output signal.
- Also we observe on CRO that by using the values 20000 and 0 to generate the high and low voltages respectively in the code, there is a voltage bias in the output signal due to internal circuit of the DSK. This unwanted bias in the output voltage can be removed by choosing such values to generate the high and low value of the voltages which are symmetric to 0. For Example, a square wave with same p-to-p voltage as the above one (i.e. 1V) can be generated by using the value 10000 to generate the minimum value of the voltage and -10000 to represent the maximum value of the voltage (Inversion!) respectively of the output signal. The output signal of this code would be symmetrical to 0V i.e. 0.5 V maximum and) 0.5 V minimum (total 1V p-to-p).
- However we observe that the bias in the output signal is constant and does not depend on the input voltage. Hence if in certain application we cannot use symmetrical values (to 0) for the high and low values of the

voltages then we can use a level shifter of constant voltage (equal to negative of the output bias) to get the correct output signal.

- If we increase the values that generate the extreme values of the voltages in the output signal in the code to +32768 and -32768 (ie. Instead of 20000 and 0 or 10000 and -10000) then there seemed to some kind of circular mapping between the DXR value and the output of D/A generator. The reason for this kind of behaviour is that we are using a 16-bit data and the maximum value that it can take is therefore $2^{16} / 2 = 32768$ (positive as well as negative). Hence the maximum p-to-p voltage that we can generate is the one corresponding to the values +32767 and
- -32767 i.e. 3V approximately.

Note: All the above observations of square waveform generation also apply to all waveform generation programs

Triangular Waveform Generation Code

```
#include <stdio.h>
#include <c6x.h>
#include "c6211dsk.h"
#include "codec_poll.h"

int main()
{
    /
        * dsp and peripheral initialization */
    CSR=0x100; /* disable all interrupts */
    IER=1; /* disable all interrupts except NMI */
    ICR=0xffff; /* clear all pending interrupts */
    *(unsigned volatile int *)EMIF_GCR = 0x3300; /* EMIF global control*/
    *(unsigned volatile int *)EMIF_CE0 = 0x30; /* EMIF CE0control */
    *(unsigned volatile int *)EMIF_CE1 = 0xffff03; /* EMIF CE1 control, 8bit async */
    *(unsigned volatile int *)EMIF_SDCTRL = 0x07117000; /* EMIF SDRAM control */
    *(unsigned volatile int *)EMIF_SDRP = 0x61a; /* EMIF SDRAM refresh period */
    *(unsigned volatile int *)EMIF_SDEXT = 0x54519; /* EMIF SDRAM extension */

    mcbbsp0_init();
    codec_playback();

    return(0);
}

void mcbbsp0_init()
{
    /* set up McBSP */

    *(unsigned volatile int *)McBSP0_SPCR = 0; /* reset serial port */
    *(unsigned volatile int *)McBSP0_PCR = 0; /* set pin control reg.*/
    *(unsigned volatile int *)McBSP0_RCR = 0x10040; /* set rx control reg. one 16 bit data/frame */
    *(unsigned volatile int *)McBSP0_XCR = 0x10040; /* set tx control reg. one 16 bit data/frame */
    *(unsigned volatile int *)McBSP0_DXR = 0;
```

```

    *(unsigned volatile int *) McBSP0_SPCR = 0x12001; /* setup SP control reg.*/
}

```

```

void mcbasp0_write(int out_data)
{
    int temp;
    temp = *(unsigned volatile int *)McBSP0_SPCR & 0x20000;
    while ( temp == 0)
    {
        temp = *(unsigned volatile int *)McBSP0_SPCR & 0x20000;
    }
    *(unsigned volatile int *)McBSP0_DXR = out_data;
}

```

```

int mcbasp0_read()
{
    int temp;
    temp = *(unsigned volatile int *)McBSP0_SPCR & 0x2;
    while ( temp == 0)
    {
        temp = *(unsigned volatile int *)McBSP0_SPCR & 0x2;
    }
    temp = *(unsigned volatile int *)McBSP0_DRR;
    return temp;
}

```

```

void codec_playback()
{

```

```

    int temp, z;
    double x[2], y;

```

```

    // set up control register 3 for S/W reset

```

```

    mcbasp0_read();
    mcbasp0_write(0);
    mcbasp0_read();
    mcbasp0_write(0);
    mcbasp0_read();
    mcbasp0_write(0);
    mcbasp0_read();
    mcbasp0_write(1);
    mcbasp0_read();
    mcbasp0_write(0x0386);
    mcbasp0_read();
    mcbasp0_write(0);

```

```

mcbasp0_read();

// set up control register 3 for mic input
mcbasp0_write(0);
mcbasp0_read();
mcbasp0_write(0);
mcbasp0_read();
mcbasp0_write(1);
mcbasp0_read();
mcbasp0_write(0x0306);
mcbasp0_read();
mcbasp0_write(0);
mcbasp0_read();

mcbasp0_write(0);
mcbasp0_read();
mcbasp0_write(1);
mcbasp0_read();
mcbasp0_write(0x2330);
temp = mcbasp0_read();
mcbasp0_write(0x0);
mcbasp0_read();
mcbasp0_write(0x0);
mcbasp0_read();
if((temp & 0xff) != 0x06)
{
#if PRINT
    printf("Error in setting up register 3.\n");
    exit(0);
#endif
}

//set up control register 4
mcbasp0_write(0);
mcbasp0_read();
mcbasp0_write(0);
mcbasp0_read();
mcbasp0_write(1);
mcbasp0_read();
mcbasp0_write(0x0400);
mcbasp0_read();
mcbasp0_write(0);
mcbasp0_read();

mcbasp0_write(0);
mcbasp0_read();

```

```

mcbasp0_write(1);
mcbasp0_read();
mcbasp0_write(0x2430);
temp = mcbasp0_read();
mcbasp0_write(0x0);
mcbasp0_read();
mcbasp0_write(0x0);
mcbasp0_read();
if((temp & 0xff) != 0x00)
{
#ifdef PRINT
    printf ("Error in setting up register 4.\n");
    exit(0);
#endif
}

// set up control register 5
mcbasp0_write(0);
mcbasp0_read();
mcbasp0_write(0);
mcbasp0_read();
mcbasp0_write(1);
mcbasp0_read();
mcbasp0_write(0x0502);
mcbasp0_read();
mcbasp0_write(0);
mcbasp0_read();

mcbasp0_write(0);
mcbasp0_read();
mcbasp0_write(1);
mcbasp0_read();
mcbasp0_write(0x2530);
temp = mcbasp0_read();
mcbasp0_write(0x0);
mcbasp0_read();
mcbasp0_write(0x0);
mcbasp0_read();
if((temp & 0xfe) != 0x2)
{
#ifdef PRINT
    printf ("Error in setting up register 5.\n");
    exit(0);
#endif
}

```

```
while(1) /* play back about 5 minutes */
{
  for(i=0;i<20000;i=i+200)
  {
    temp = mcbsp0_read();
    temp = i;
    temp = temp & 0xfffe;
    mcbsp0_write(temp);
  }
  for(i=20000;i>0;i=i-200)
  {
    temp = mcbsp0_read();
    temp = i;
    temp = temp & 0xfffe;
    mcbsp0_write(temp);
  }
}
}
```

Some Important Observations about the Triangular Waveform Generation

Code

Manipulation of the amplitude and frequency of the output signal :

We observe that the frequency of the output signal would depend on the number of times the loop containing “i” runs.

How long the loop will continue will still depend on two things. The loop can be made to continue longer either by increasing the maximum value that “ i ” takes while keeping the increment in “ i ” constant OR by decreasing the increment in “ i ” while keeping the maximum value of “ i ” constant.

This is because the higher the maximum value of i (keeping the increment of “ i ” constant) , the voltage of the output triangular signal would continue increasing till a higher value (corresponding to higher maximum value of “ i ”) , which it will take more time and hence the frequency of the output signal would decrease.

Similarly, if we decrease the increment in “ i ” (keeping the maximum value of the maximum value of “ i ” constant) the loop will continue more number of times to reach the same maximum value of “ i ”.

The amplitude of the wave does not depend on the increment in the value of “ i ” but on the maximum value that “ i ” takes. Hence in the latter case the amplitude of the wave would remain the same and only the frequency would decrease.

However in the former case, the amplitude of the wave would also increase (because the voltage of the output signal would increase to a higher value (corresponding to the new maximum value of i) as compared to earlier

maximum value of i). If we want that the amplitude remain constant while changing the frequency, we should multiply the argument of the `mcbasp_write()` function by the required factor.

Sawtooth Waveform Generation Code

```
#include <stdio.h>
#include <c6x.h>
#include "c6211dsk.h"
#include "codec_poll.h"

int main()
{

    /* dsp and peripheral initialization */
    CSR=0x100;                /* disable all interrupts */
    IER=1;                    /* disable all interrupts except NMI */
    ICR=0xffff;              /* clear all pending interrupts */

    *(unsigned volatile int *)EMIF_GCR = 0x3300; /* EMIF global control*/
    *(unsigned volatile int *)EMIF_CE0 = 0x30; /* EMIF CE0control */
    *(unsigned volatile int *)EMIF_CE1 = 0xffff03;
    /* EMIF CE1 control, 8bit async */
    *(unsigned volatile int *)EMIF_SDCTRL = 0x07117000;
    /* EMIF SDRAM control */
    *(unsigned volatile int *)EMIF_SDRP = 0x61a;
    /* EMIF SDRAM refresh period */
    *(unsigned volatile int *)EMIF_SDEXT = 0x54519;
    /* EMIF SDRAM extension */

    mcbbsp0_init();
    codec_playback();

    return(0);
}

void mcbbsp0_init()
{
    /* set up McBSP */

    *(unsigned volatile int *)McBSP0_SPCR = 0;
    /* reset serial port */
    *(unsigned volatile int *)McBSP0_PCR = 0;
    /* set pin control reg.*/
    *(unsigned volatile int *)McBSP0_RCR = 0x10040;
```

```

    /* set rx control reg. one 16 bit data/frame */
    *(unsigned volatile int *)McBSP0_XCR = 0x10040;
    /* set tx control reg. one 16 bit data/frame */
    *(unsigned volatile int *)McBSP0_DXR = 0;
    *(unsigned volatile int *)McBSP0_SPCR = 0x12001;
    /* setup SP control reg.*/
}

void mcbasp0_write(int out_data)
{
    int temp;
    temp = *(unsigned volatile int *)McBSP0_SPCR & 0x20000;
    while ( temp == 0)
    {
        temp = *(unsigned volatile int *)McBSP0_SPCR & 0x20000;
    }
    *(unsigned volatile int *)McBSP0_DXR = out_data;
}

int mcbasp0_read()
{
    int temp;
    temp = *(unsigned volatile int *)McBSP0_SPCR & 0x2;
    while ( temp == 0)
    {
        temp = *(unsigned volatile int *)McBSP0_SPCR & 0x2;
    }
    temp = *(unsigned volatile int *)McBSP0_DRR;
    return temp;
}

void codec_playback()
{
    int temp, z;
    double x[2], y;

    // set up control register 3 for S/W reset
    mcbasp0_read();
    mcbasp0_write(0);
    mcbasp0_read();
    mcbasp0_write(0);
    mcbasp0_read();
    mcbasp0_write(0);
    mcbasp0_read();
}

```

```

mcbasp0_write(1);
mcbasp0_read();
mcbasp0_write(0x0386);
mcbasp0_read();
mcbasp0_write(0);
mcbasp0_read();

// set up control register 3 for mic input
mcbasp0_write(0);
mcbasp0_read();
mcbasp0_write(0);
mcbasp0_read();
mcbasp0_write(1);
mcbasp0_read();
mcbasp0_write(0x0306);
mcbasp0_read();
mcbasp0_write(0);
mcbasp0_read();

mcbasp0_write(0);
mcbasp0_read();
mcbasp0_write(1);
mcbasp0_read();
mcbasp0_write(0x2330);
temp = mcbasp0_read();
mcbasp0_write(0x0);
mcbasp0_read();
mcbasp0_write(0x0);
mcbasp0_read();
if((temp & 0xff) != 0x06)
{
#ifdef PRINT
    printf ("Error in setting up register 3.\n");
    exit(0);
#endif
}

//set up control register 4
mcbasp0_write(0);
mcbasp0_read();
mcbasp0_write(0);
mcbasp0_read();
mcbasp0_write(1);
mcbasp0_read();
mcbasp0_write(0x0400);
mcbasp0_read();

```

```

mcbasp0_write(0);
mcbasp0_read();

mcbasp0_write(0);
mcbasp0_read();
mcbasp0_write(1);
mcbasp0_read();
mcbasp0_write(0x2430);
temp = mcbasp0_read();
mcbasp0_write(0x0);
mcbasp0_read();
mcbasp0_write(0x0);
mcbasp0_read();
if((temp & 0xff) != 0x00)
{
#ifdef PRINT
    printf("Error in setting up register 4.\n");
    exit(0);
#endif
}

// set up control register 5
mcbasp0_write(0);
mcbasp0_read();
mcbasp0_write(0);
mcbasp0_read();
mcbasp0_write(1);
mcbasp0_read();
mcbasp0_write(0x0502);
mcbasp0_read();
mcbasp0_write(0);
mcbasp0_read();

mcbasp0_write(0);
mcbasp0_read();
mcbasp0_write(1);
mcbasp0_read();
mcbasp0_write(0x2530);
temp = mcbasp0_read();
mcbasp0_write(0x0);
mcbasp0_read();
mcbasp0_write(0x0);
mcbasp0_read();
if((temp & 0xfe) != 0x2)
{
#ifdef PRINT

```

```
    printf ("Error in setting up register 5.\n");
    exit(0);
#endif
}

while(1) /* play back about 5 minutes */
{
    for(i=0;i<10000;i=i+200)
    {
        temp = i;
        temp = temp & 0xfffe;
        mcbasp0_write(temp);
    }
}
}
```

All the observations about the square and the triangular waveform generation apply to the sawtooth waveform generation also.

Sampling

```
#include <stdio.h>
#include <c6x.h>
#include "c6211dsk.h"
#include "codec_poll.h"

int main()
{
    /          * dsp and peripheral initialization */
    CSR=0x100; /* disable all interrupts */
    IER=1; /* disable all interrupts except NMI */
    ICR=0xffff; /* clear all pending interrupts */
    *(unsigned volatile int *)EMIF_GCR = 0x3300; /* EMIF global control*/
    *(unsigned volatile int *)EMIF_CE0 = 0x30; /* EMIF CE0control */
    *(unsigned volatile int *)EMIF_CE1 = 0xffff03; /* EMIF CE1 control, 8bit async */
    *(unsigned volatile int *)EMIF_SDCTRL = 0x07117000; /* EMIF SDRAM control */
    *(unsigned volatile int *)EMIF_SDRP = 0x61a; /* EMIF SDRM refresh period */
    *(unsigned volatile int *)EMIF_SDEXT = 0x54519; /* EMIF SDRAM extension */

    mcbbsp0_init();
    codec_playback();

    return(0);
}

void mcbbsp0_init()
{
    /* set up McBSP */

    *(unsigned volatile int *)McBSP0_SPCR = 0; /* reset serial port */
    *(unsigned volatile int *)McBSP0_PCR = 0; /* set pin control reg.*/
    *(unsigned volatile int *)McBSP0_RCR = 0x10040; /* set rx control reg. one 16 bit data/frame */
    *(unsigned volatile int *)McBSP0_XCR = 0x10040; /* set tx control reg. one 16 bit data/frame */
    *(unsigned volatile int *)McBSP0_DXR = 0;
    *(unsigned volatile int *)McBSP0_SPCR = 0x12001; /* setup SP control reg.*/
}
```

```

}

void mcbasp0_write(int out_data)
{
    int temp;
    temp = *(unsigned volatile int *)McBSP0_SPCR & 0x20000;
    while ( temp == 0)
    {
        temp = *(unsigned volatile int *)McBSP0_SPCR & 0x20000;
    }
    *(unsigned volatile int *)McBSP0_DXR = out_data;
}

int mcbasp0_read()
{
    int temp;
    temp = *(unsigned volatile int *)McBSP0_SPCR & 0x2;
    while ( temp == 0)
    {
        temp = *(unsigned volatile int *)McBSP0_SPCR & 0x2;
    }
    temp = *(unsigned volatile int *)McBSP0_DRR;
    return temp;
}

void codec_playback()
{
    int temp, z;
    double x[2], y;

    // set up control register 3 for S/W reset
    mcbasp0_read();
    mcbasp0_write(0);
    mcbasp0_read();
    mcbasp0_write(0);
    mcbasp0_read();
    mcbasp0_write(0);
    mcbasp0_read();
    mcbasp0_write(1);
    mcbasp0_read();
    mcbasp0_write(0x0386);
    mcbasp0_read();
    mcbasp0_write(0);
    mcbasp0_read();
}

```

```

// set up control register 3 for mic input
mcbasp0_write(0);
mcbasp0_read();
mcbasp0_write(0);
mcbasp0_read();
mcbasp0_write(1);
mcbasp0_read();
mcbasp0_write(0x0306);
mcbasp0_read();
mcbasp0_write(0);
mcbasp0_read();

mcbasp0_write(0);
mcbasp0_read();
mcbasp0_write(1);
mcbasp0_read();
mcbasp0_write(0x2330);
temp = mcbasp0_read();
mcbasp0_write(0x0);
mcbasp0_read();
mcbasp0_write(0x0);
mcbasp0_read();
if((temp & 0xff) != 0x06)
{
#ifdef PRINT
    printf ("Error in setting up register 3.\n");
    exit(0);
#endif
}

//set up control register 4
mcbasp0_write(0);
mcbasp0_read();
mcbasp0_write(0);
mcbasp0_read();
mcbasp0_write(1);
mcbasp0_read();
mcbasp0_write(0x0400);
mcbasp0_read();
mcbasp0_write(0);
mcbasp0_read();

mcbasp0_write(0);
mcbasp0_read();
mcbasp0_write(1);

```

```

mcbasp0_read();
mcbasp0_write(0x2430);
temp = mcbasp0_read();
mcbasp0_write(0x0);
mcbasp0_read();
mcbasp0_write(0x0);
mcbasp0_read();
if((temp & 0xff) != 0x00)
{
#ifdef PRINT
    printf ("Error in setting up register 4.\n");
    exit(0);
#endif
}

// set up control register 5
mcbasp0_write(0);
mcbasp0_read();
mcbasp0_write(0);
mcbasp0_read();
mcbasp0_write(1);
mcbasp0_read();
mcbasp0_write(0x0502);
mcbasp0_read();
mcbasp0_write(0);
mcbasp0_read();

mcbasp0_write(0);
mcbasp0_read();
mcbasp0_write(1);
mcbasp0_read();
mcbasp0_write(0x2530);
temp = mcbasp0_read();
mcbasp0_write(0x0);
mcbasp0_read();
mcbasp0_write(0x0);
mcbasp0_read();
if((temp & 0xfe) != 0x2)
{
#ifdef PRINT
    printf ("Error in setting up register 5.\n");
    exit(0);
#endif
}
while(1) /* play back about 5 minutes */
{

```

```
temp = mcbsp0_read();  
temp = temp & 0xffe;  
mcbsp0_write(temp);  
}  
}
```

The Sampling frequency can be measured by changing the frequency of the input signal and observing the output signal. However the sampling frequency is an inbuilt property of the DSK (since it depends on the sampling frequency of the A/D (AD 535) converter (~8 kHz) which is there on the board itself) and hence the sampling frequency can not be changed.

Filters

In the codes for the filters , the initialization of registers and McBSP is exactly the same as in the waveform generation. The only addition is that of the transfer function $h[n]$ and the only difference is in the actual code (processing part).

We shall outline only the changes in the code below.

Finite Impulse Response

- Order - 8 High Pass

```
/*Sample Transfer Function*/
```

```
double h[8] = { 0.008987, 0.068136, 0.06694, -0.490672, 0.490672, -0.06694, -0.068136, -0.008987 };
```

```
/*Processing the data*/
```

```
while(1) /* play back about 5 minutes */
```

```
{
    x[0] = mcbasp0_read();
    y = 0;
    for(i=0; i<8; i++)
        y = y + x[i] * h[7-i];
    z = y;
    z = z & 0xfffe;
    mcbasp0_write(z);
    for(i=6; i>=0; i--)
    {
        x[i+1] = x[i];
    }
}
```

- Order – 8 Low Pass

```
/*Sample Transfer Function*/
```

```
double h[8] = { 0.008987, -0.068136, 0.06694, 0.490672, 0.490672, 0.06694, -0.068136, 0.008987 };
```

```
/*Processing the data*/
```

```
while(1) /* play back about 5 minutes */
```

```
{
    x[0] = mcbasp0_read();
    y = 0;
    for(i=0; i<8; i++)
```

```
    y = y + x[i] * h[7-i];
z = y;
z = z & 0xfffe;
mcbasp0_write(z);
for(i=6; i>=0; i--)
{
    x[i+1] = x[i];
}
}
```

- Simple High Pass

```
/*Processing the data*/
while(1) /* play back about 5 minutes */
{
    x[0] = mcbasp0_read();
    y = ( x[0] - x[1] ) / 2;
    z = y;
    z = z & 0xfffe;
    mcbasp0_write(z);
    x[1] = x[0];
}
```

- Simple Low Pass

```
/*Processing the data*/
while(1) /* play back about 5 minutes */
{
    x[0] = mcbasp0_read();
    y = ( x[0] + x[1] ) / 2;
    z = y;
    z = z & 0xfffe;
    mcbasp0_write(z);
    x[1] = x[0];
}
```

Infinite Impulse Response

- Simple High pass

```
/*Processing the data*/
while(1) /* play back about 5 minutes */
{
    x[0] = mcbsp0_read();
    y = ( x[0] - x[1] ) / 2 + y/2;
    z = y;
    z = z & 0xfffe;
    mcbsp0_write(z);
    x[1] = x[0];
}
```

- Simple Low Pass

```
/*Processing the data*/
while(1) /* play back about 5 minutes */
{
    x[0] = mcbsp0_read();
    y = ( x[0] + x[1] ) / 2 - y/8;
    z = y;
    z = z & 0xfffe;
    mcbsp0_write(z);
    x[1] = x[0];
}
```
